CORE JAVA MODULE 6 NOTES

01 July 2025 15:43

➢ Module 6: Object-Oriented Programming (OOP)

No.	Topic
6.1	Class and Object in Java
6.2	Constructors – Default, Parameterized, Copy (Ways to initialize the objects)
6.3	'this' Keyword and Constructor Chaining
6.4	Inheritance – Single, Multilevel, Hierarchical
6.5	Method Overriding
6.6	Polymorphism (Compile Time & Run Time)
6.7	Abstraction – Abstract Class & Interface
6.8	Encapsulation – private + public getters/setters
6.9	Access Modifiers – public, private, protected, default
6.10	'super' Keyword
6.11	final keyword – variable, method, class
6.12	Static Keyword – static variable, method, block
6.13	Inner Classes – Normal, Static, Anonymous
6.14	Interview Questions and Assignment

⇒ Module : 6.1 – Class and Object in Java (with Real-Life Analogy)

- **First, Tell me :**

Why do we even need OOP when we already had procedural programming?"

- **Real-World Problem**

Let's say you are building a **school management system**.
In procedural style:

```
String student1Name = "Riya";
int student1Age = 16;

String student2Name = "Aryan";
int student2Age = 17;
```

What if you had 1000 students?
What if you also want to store marks, class, roll number, address?

It becomes **messy**, repetitive, and hard to manage.

◆ Real-World Analogy: Classroom & Blueprints

Imagine:

- You're an **architect** designing houses .
- You first create a **blueprint** .
- From that blueprint, you can build **many houses**.
  Blueprint = Class

Actual House = Object (Instance of the class)

Same in Java:

```
// Class (Blueprint)
class Student {
    String name;
    int age;
}

// Object (Real Entity)
Student s1 = new Student();  // House 1
Student s2 = new Student();  // House 2
```

- Object oriented programming system / structure
- OOP is a programming paradigm/methodology (way of doing work)

Example 1: *"You want to travel."*

Real-life: You can go by car, bike, train, or airplane.
OOP View:

- **TravelMode** → Class
- **car, bike, train** → Objects
- **go()** → Method
- **speed, cost, comfort** → Properties

Each object performs the same action (go()), but differently — that's **polymorphism**, which comes later in OOP.

Example 2: *"A person wants to earn money."*

Real-life: A person can choose different ways — job, freelancing, business, etc.
OOP View:

- **Person** → Class
- **job(), business()** → Methods
- **name, age, skill** → Properties

Each person (object) can take different paths based on their attributes (variables) and decisions (methods).

6 main Piller of OOP's:

1. Class
2. Object & Method
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

✧ **Class:**
-> Real world ex: Animal , Vehicle , fruits
So these are classes

✧ **Object:**
Real world ex:
-> Animal --------> Dog , Cat , tiger , lion

-> Vehicle -------> Car , Bike
->Fruits ----------> Apple , Banana

✧ **Method:**

Method is just like a task which we perform
Real life ex:
Animal (eat) -> eat is just like a task
Animal (run) -> run is just like a task

How these method work :
Like who will eat -> dog (so we will call the method with the help of object)

Now we will relate all these concepts through programming.

✧ What is a Class?

- A **class** is a **user-defined data type** that groups related data and behavior.
- It defines **properties** (variables) and **actions** (methods).
- Class is not the real world entity , it is just like blueprint .
- Class does not occupy memory.

Java Syntax:

```
class Student {
    // Data
    String name;
    int age;

    // Behavior
    void study() {
        System.out.println(name + " is studying");
    }
}
```

✧ What is an Object?

- An **object** is the **real-world instance** of a class.
- It has its **own copy** of variables and can call the methods of the class.

- **Syntax: Object Creation in Java**

ClassName objectName = new ClassName();

- **How to create an object:**

```
Student s1 = new Student();
```

⇒ Module 6.2: Constructors – Default, Parameterized, Copy

First: Why Do We Even Need Constructors?

Q : "When you create an object with new, how does the object get values?"

```
Student s1 = new Student();
s1.name = "Aman";
s1.age = 20;
```

"This works, but what if I want to set all values **automatically** when I create the object?"
That's where **constructors** come into the picture.

Real-Life Analogy:

Cake Factory Analogy:

- **Class** = Cake Factory
- **Object** = A cake
- **Constructor** = The machine that **initializes** the cake with flavor, size, price

When you say:

```
Cake chocolate = new Cake("Chocolate", 500);
```

"Hey machine! Give me a chocolate cake with price ₹500."

So in Java:

```
Student s1 = new Student("Aman", 20);
```

What is a Constructor?

A **constructor** is a special method that:

- Has **same name as the class**
- Has **no return type (not even void)**
- Is **called automatically** when an object is created
- Is used to **initialize** object data (variables)

→ 3 Types of Constructors

1. Default Constructor

```
class Student {
    String name;
    int age;

    // Default constructor
    Student() {
        System.out.println("Constructor called");
    }
}


Student s1 = new Student(); //  constructor runs automatically
```

2. Parameterized Constructor

```
class Student {
    String name;
    int age;

    Student(String n, int a) {
        name = n;
        age = a;
    }

    void show() {
```

```
        System.out.println(name + " : " + age);
    }
}
```

3. Copy Constructor (Manual in Java)

```
class Student {
    String name;
    int age;

    Student(String n, int a) {
        name = n;
        age = a;
    }

    // Copy constructor
    Student(Student s) {
        name = s.name;
        age = s.age;
    }
}

Student s1 = new Student("Ravi", 22);
Student s2 = new Student(s1); //  Copy of s1
```

Why Are Constructors Important?

Reason	Why it matters
◆ Automatic Initialization	◆ Avoids setting each field manually
◆ Cleaner Code	◆ One line creates + sets values
◆ Reusability	◆ Can reuse logic for setting up objects
◆ Overloading	◆ Can create multiple versions with different input types

✧ Ways to Initialize Objects in Java

Real-World Analogy First:

Let's say you buy a new **laptop**. How do you set it up?

Way	Analogy
◆ Dot notation	◆ You manually install apps one by one
◆ Constructor	◆ It comes pre-installed and ready-to-go
◆ Method	◆ A technician sets it up later for you

Same way in Java — an object needs values. You can give those values in **3 ways**

1. Using Dot Notation (Reference Variable)

```
Student s1 = new Student();
s1.name = "Aman";
s1.age = 20;
```

✅ Pros	❌ Cons
◆ Easy to understand	◆ No validation
◆ Good for beginners/testing	◆ Can forget to set some values

- Useful for simple objects
- Long code for big objects

2. Using Constructor

Student s2 = new Student("Aman", 20);

```
class Student {
    String name;
    int age;
    Student(String n, int a) {
        name = n;
        age = a;
    }
}
```

✅ Pros	❌ Cons
Best for setting values during object creation	Fixed number/order of arguments
Cleaner and shorter code	Can get confusing with too many args
Supports constructor overloading	No logic allowed after creation

3. Using Methods (e.g., setData() or setters)

```
Student s3 = new Student();
s3.setData("Aman", 20);

void setData(String name, int age) {
    this.name = name;
    this.age = age;
}
```

✅ Pros	❌ Cons
Can add validation	Requires extra step after object is created
Can call multiple times	May forget to call it
Very flexible	Might lead to inconsistent object

So, Which Way is Recommended?

Situation	Recommended Way
Quick testing / simple value setting	Dot operator (manual)
Professional projects / clean code	Constructor
Need dynamic / late setting	Method (setData() / setters)
Copy same object	Copy constructor

PROGRAM :

```
class Student {
    String name;
    int age;

    // 1. Default / Non-Parameterized Constructor
    Student() {
```

```
        System.out.println("Default constructor called");
        name = "Not Assigned";
        age = 0;
    }

    // 2. Parameterized Constructor
    Student(String name, int age) {
        System.out.println("Parameterized constructor called");
        this.name = name;
        this.age = age;
    }

    // 3. Copy Constructor
    Student(Student s) {
        System.out.println("Copy constructor called");
        this.name = s.name;
        this.age = s.age;
    }

    // 4. Method to set data manually
    void setData(String name, int age) {
        System.out.println("setData() method called");
        this.name = name;
        this.age = age;
    }

    // Method to display student details
    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
        System.out.println("------------------------------");
    }
}

public class Main {
    public static void main(String[] args) {

        // 1 Way 1: Dot Notation (reference variable)
        System.out.println("=== Way 1: Dot Notation ===");
        Student s1 = new Student();
        s1.name = "Aman";
        s1.age = 20;
        s1.display();

        // 2 Way 2: Parameterized Constructor
        System.out.println("=== Way 2: Parameterized Constructor ===");
        Student s2 = new Student("Riya", 22);
        s2.display();

        // 3 Way 3: Method (setData)
        System.out.println("=== Way 3: Method (setData) ===");
        Student s3 = new Student(); // Default constructor will run
        s3.setData("Karan", 21);
        s3.display();

        // 4 Way 4: Copy Constructor
        System.out.println("=== Way 4: Copy Constructor ===");
        Student s4 = new Student(s2); // Copying s2
        s4.display();
    }
}
```

Output:

```
=== Way 1: Dot Notation ===
Default constructor called
Name: Aman, Age: 20
        ----------------------------
        === Way 2: Parameterized Constructor ===
Parameterized constructor called
Name: Riya, Age: 22
        ----------------------------
        === Way 3: Method (setData) ===
Default constructor called
setData() method called
Name: Karan, Age: 21
        ----------------------------
        === Way 4: Copy Constructor ===
Copy constructor called
Name: Riya, Age: 22
        ----------------------------
```

✧ Module : 6.3 – this Keyword and Constructor Chaining

◇ Real-Life Analogy (Student-Friendly)

- Imagine your friend group has two people named **"Rahul"**.
- If you say "Rahul, pass me the pen" — which one will respond?You'll need to say:
- **"This Rahul in front of me"**

- Same in Java!
- When a **method/constructor** receives variables with the **same name** as instance variables, Java gets **confused**.
- We use this to say:
  - "This object's variable, not the local one."

◇ What is this Keyword?

this is a **reference to the current object**

We use it to:

Use Case	Example
Distinguish between instance & local vars	this.name = name;
Call another constructor	this(...);
Return current object	return this;
Pass current object as argument	someMethod(this);

EXMAPLE :

```
class Student {
   String name;
   int age;

   // 1 this() - Constructor Chaining
   Student() {
      this("Unknown", 0);  // Calling parameterized constructor
      System.out.println("Default constructor called");
   }
```

```
    // [2] this.variable - To resolve variable shadowing
    Student(String name, int age) {
        this.name = name;  // this.name = instance variable, name = parameter
        this.age = age;
        System.out.println("Parameterized constructor called");
    }

    // [3] this.method() - Calling a method from another method
    void displayInfo() {
        this.printGreeting(); // calls method from same class
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }

    void printGreeting() {
        System.out.println("Hello from printGreeting method!");
    }

    // [4] this as argument
    void sendToPrinter() {
        Printer.printStudent(this);  // passing current object
    }

    // [5] return this - Method chaining
    Student updateName(String name) {
        this.name = name;
        return this; // returns current object
    }

    Student updateAge(int age) {
        this.age = age;
        return this;
    }
}

class Printer {
    static void printStudent(Student s) {
        System.out.println("Printer -> Student Name: " + s.name + ", Age: " + s.age);
    }
}

public class ThisKeywordDemo {
    public static void main(String[] args) {
        // Calls default constructor -> which calls parameterized using this()
        Student s1 = new Student();
        s1.displayInfo();
        s1.sendToPrinter();

        System.out.println("-----------------------");

        // Direct object with parameterized constructor
        Student s2 = new Student("Rahul", 21);
        s2.displayInfo();
        s2.sendToPrinter();

        System.out.println("-----------------------");

        // [5] Method chaining using return this
```

```
        Student s3 = new Student().updateName("Meena").updateAge(18);
        s3.displayInfo();
    }
}
```

⇒Module : 6.4 – INHERITANCE IN JAVA:

Real-Life Analogy

- **Imagine a Family:**
  - 👴 Grandpa → 👨 Father → 👦 Son
  - Father inherits from Grandpa
  - Son inherits from Father
    → That's **Multilevel Inheritance**.

Now imagine:

- One 👨 father → Two kids: 👦 Son1, 👧 Daughter
  → That's **Hierarchical Inheritance**.

- ### What is Inheritance?

  Inheritance allows one class to **acquire (inherit) the properties and methods** of another class.

  ◈ Why Use It?
  - **Code Reusability** (don't repeat)
  - **Extensibility**
  - **Logical Hierarchy**

  ◈ Syntax

```
class Parent {
    // parent properties & methods
}

class Child extends Parent {
    // child-specific properties & methods
}
```

➢ Types of Inheritance (That Java Supports)

1 Single Inheritance

```
class Animal {
    void eat() {
        System.out.println("I can eat");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("I can bark");
    }
}
```

2 Multilevel Inheritance

Child → Parent → Grandparent

```
class Animal {
    void eat() {
        System.out.println("I can eat");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("I can bark");
    }
}

class Puppy extends Dog {
    void weep() {
        System.out.println("I can weep");
    }
}
```

3 Hierarchical Inheritance

One parent → many children

```
class Animal {
    void eat() {
        System.out.println("I can eat");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("I can bark");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("I can meow");
    }
}
```


Not Supported Directly:

4 Multiple Inheritance (with classes)

```
class A {}
class B {}
class C extends A, B {} // ✕ Error
```

Java does **not allow** this to avoid **diamond problem**. Use **interfaces** instead.

Q:Why Multiple Inheritance is not allowed in Java with classes?

Multiple Inheritance = A class inherits from **more than one class**.

```
class A {
```

```
    void show() {
        System.out.println("From A");
    }
}

class B {
    void show() {
        System.out.println("From B");
    }
}

// ❌ This is NOT allowed in Java
class C extends A, B {
    // Which show() should be called??
}
```

Problem: **Diamond Problem**

```
  A
 / \
B   C
 \ /
  D
```

- If both B and C inherit from A, and D inherits from both B and C,
- now D gets 2 copies of A's methods. So…
  - Q: Which version of A's method should D inherit?
  - Ambiguity!
- That's why Java **does not allow** multiple inheritance using classes.

Real-Life Analogy

Imagine you have 2 bosses:

- Boss A says: "Work from home."
- Boss B says: "Come to office."

Which one do you obey?
That's the **Diamond Problem**. Ambiguity

⇒ Module : 6.6 POLYMORPHISM (Compile-Time vs Run-Time)

- **Poly** = many, **Morphism** = forms (ek naam, multiple kaam)
- Polymorphism allows one entity (method or object) to take multiple forms, enabling flexibility and code reusability.

Types:

Type	Also Called	Example	Happens When?
Compile-Time	Method Overloading	Same method name, diff params	During compilation
Run-Time	Method Overriding	Same method name, same params	During execution

✧ Compile-Time Polymorphism: Method Overloading(Compiler handle this):

- **WhatsApp Send Feature**

```
class WhatsApp {
    void sendMessage(String text) {
        System.out.println("Text: " + text);
```

```
    }

    void sendMessage(String text, String emoji) {
        System.out.println("Text with Emoji: " + text + " " + emoji);
    }

    void sendMessage(byte[] imageData) {
        System.out.println("Sending an image...");
    }
}
```

- *Method sendMessage() is **overloaded** with different types of parameters (String, byte[])*

Example : Bank ATM Transactions, *Amazon Delivery System*

✧ **Method Overriding (Run-Time Polymorphism)**

What is Method Overriding?

Method in **child class** has **same name, return type, and parameters** as a method in **parent class**, but **different body** (custom logic).

- Happens at **runtime**
- Enables **runtime polymorphism / dynamic method dispatch**

Syntax Rules:

Rule

- Method name must be **same**
- Parameters must be **same**
- Return type must be **same or covariant**
- Child class must use **@Override** (optional, but recommended)
- Only works in **inheritance** (extends)

Example :

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}
```

Main Method (Run-Time Polymorphism):

```
public class OverrideDemo {
    public static void main(String[] args) {
        Animal a1 = new Dog();  // Animal reference → Dog object
        Animal a2 = new Cat();  // Animal reference → Cat object

        a1.sound();  // Output: Dog barks
        a2.sound();  // Output: Cat meows
    }
}
```

Why? Because at **runtime**, Java decides the actual object type (Dog or Cat) and calls the **overridden method**.

Method Overriding vs Overloading:

Feature	Overloading	Overriding
◆ Based on	◆ Different parameters	◆ Same method, redefined
◆ Happens at	◆ Compile time	◆ Runtime
◆ Class requirement	◆ Same class	◆ Inheritance (Parent → Child)
◆ Return type	◆ Can differ	◆ Must be same (or covariant)
◆ Example	◆ sendMessage(text) + sendMessage(image)	◆ sound() in Animal, Dog, Cat

◆ **Why This Is Powerful (Real-Life Example):**

Ola/Uber Example:

- You write: Vehicle vehicle = new Ride();
- Today, Ride is a Bike
- Tomorrow, it could be a Car, Auto, or Truck

You don't have to change your business logic:

⇒ Module 6.7 – Abstraction (Abstract Class & Interface)

✧ What is Abstraction?

Hiding internal implementation and showing only **essential features**.
Just like:

Real-Life Example:

You drink juice, but you don't care how it's made inside the machine.

- You just press a button: Juice comes out.
- You don't know about the motor, blades, pressure — that's **hidden**.

That's **Abstraction** — only **important things are exposed**, rest is hidden.

- **ATM Machine** – you only see the screen & buttons, not the wiring inside.
- **Car** – you press the accelerator, but don't care how the engine works.

- In Java:

- We hide *how* things work and just show *what* they do — using:
- abstract class
- interface

Real-Life Analogy

Imagine a remote-controlled toy:

- You press a **button** → Toy moves
- You don't know **how internally it works**

➤ You're only interacting with the **interface (buttons)**, not the internal mechanics.

Same in Java:

```
// You just know what should happen
void start();
void stop();
```

You **don't care how** it's coded inside.
That's abstraction!

Abstraction in OOP with Java:

In Java, abstraction can be achieved using two main approaches:

1. **Abstract Classes(0-100%) 10% 50% 100% 0%**
2. **Interfaces(100%)**

Why Use Abstraction?

Reason	Benefit
◆ Hide complexity	◆ Code is cleaner
◆ Focus on *what*, not *how*	◆ Makes development easier
◆ Force subclasses to implement logic	◆ Helps maintain consistency
◆ Support multiple developers	◆ One writes structure, others write implementation

1. Abstract Class (Partial Abstraction)

- An **abstract class** in Java is a class that cannot be instantiated directly.
- It can have abstract methods (methods without implementation) that must be implemented by its subclasses. An abstract class may also have concrete methods (methods with implementation).

A) A method without body known as abstract method.
B) If a class has an abstract method then class should be declared as a abstract.
C) In abstract class I can create both abstract method and simple method(concrete) that why (0-100%)abstraction achieve in this .
D) If a sub-class extends an abstract class, it is compulsory to implement the body of that abstract method in subclass

Main point to remember : We can't create the object of abstract class , but we can create reference .

```
abstract class Vehicle {
    abstract void start();  // Unimplemented

    void fuel() {          // Implemented
```

```
        System.out.println("Fueling vehicle");
    }
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car starts with key");
    }
}

public class Test {
    public static void main(String[] args) {
        Vehicle v = new Car();  // Parent reference
        v.start();              // Car starts with key
        v.fuel();               // Fueling vehicle
    }
}
```

Key Points:

Abstract Class:

1. **Allowed:**
   - Abstract methods (without body).
   - Concrete methods (with body).
   - Instance variables (non-static fields).
   - Static variables.
   - Static methods (from Java 8 onwards).
   - Constructors.
   - Final methods (cannot be overridden).
   - Access modifiers (like public, protected, private).
2. **Not Allowed:**
   - Object instantiation (cannot create objects of an abstract class).

Q:Why Concrete Methods in Abstract Class?

```
abstract class Animal {
    abstract void sound(); // abstract => must be implemented

    public void sleep() {
        System.out.println("Sleeping...");
    }
}
```

So:

- sound() is **abstract** — subclass **must** define it.
- sleep() is **concrete** — subclass can use it as-is.

But wait... isn't this breaking abstraction?
No! It's enhancing it. Why?
Because:

- sleep() is a **common behavior** — every animal sleeps the same way (no need to override).
- sound() is a **variable behavior** — Dog and Cat have different sounds, so we abstract that. This is called **"partial abstraction"**.

Use-Case for Concrete Methods in Abstract Class:

Let's say you're creating a PaymentGateway:

```
abstract class PaymentGateway {

    // common step for all payments
    public void connectToServer() {
        System.out.println("Connected to bank server");
    }

    // different for each payment type
    abstract void pay(int amount);
}

class UPI extends PaymentGateway {
    void pay(int amount) {
        System.out.println("Paid via UPI: ₹" + amount);
    }
}
```

Now in the main :

```
PaymentGateway g = new UPI();
g.connectToServer(); // same for all
g.pay(500);          // specific for UPI
```

Advantage:

- Common code (connectToServer) is **centralized** and **reused**.
- Variable code (pay) is **forced to be customized** by each subclass.

✧ Interface (100% Abstraction before Java 8)

Interface is a contract that defines what should be done, not how.

Real-Life Analogy:

Think of a remote control.

- Remote has buttons: powerOn(), volumeUp(), etc. — **interface**
- TV, AC, Projector — each device **implements** those buttons differently.

➞ Interface = **Button labels**

➞ Class = **Device that responds differently.**

A) We can create only abstract method in the interface.
B) Interface is the blue print of the class. It specify what a class must do not how
C) It is used to achieve abstraction
D) It support multiple inheritance
E) We can't create object of interface

```
SYNTAX : interface interfaceName{

// methods (only abstract method) remember that it should be public
// field (public static final)

    void show(); // by default this is public & abstract:

}
```

Conclusion: When to Use an Interface?

- When **multiple classes** need to follow the same behavior but implement it differently.
- When you want to **achieve multiple inheritance** in Java.

Interface:

1. **Allowed:**
   - Abstract methods (implicitly public and abstract).
   - Default methods (with body, using default keyword, introduced in Java 8).
   - Static methods (with body, introduced in Java 8).
   - Constant variables (implicitly public, static, and final).
2. **Not Allowed:**
   - Instance variables (non-static fields).
   - Constructors (cannot create objects of an interface).
   - Final methods (because all methods are abstract by default unless default or static is used).

```
interface Remote {
    void pressPower();  // Only method signatures (no body)
}

class TV implements Remote {
    public void pressPower() {
        System.out.println("TV turned ON");
    }
}

public class Main {
    public static void main(String[] args) {
        Remote r = new TV();
        r.pressPower();   // Output: TV turned ON
    }
}
```

Abstract Class vs Interface – Ultimate Table

Feature	Abstract Class	Interface
Use	Partial abstraction	Full abstraction
Methods	Can have both abstract + concrete	Only abstract (Java <8)
Inheritance	extends	implements
Multiple inheritance	No	Yes
Variables	Can have instance vars	Only static final (constants)
Access Modifiers	public, protected, private	Only public
Real-life analogy	Template with partial design	Contract or rulebook

Interface is best when:

- You want to **enforce rules**
- You want **unrelated classes** to follow common behavior
- You need **multiple inheritance**
- You care about **flexibility**, not code reuse

Like:

```
interface Payable {
    void makePayment();
}
class Employee implements Payable {}
```

```
class Freelancer implements Payable {}
```

Employee and Freelancer are totally different — but both can "makePayment"

Abstract class is best when:

- You want to **share logic** (like a base class)
- You have a **common family of related classes**
- You want to write **partial implementation**
- You want to define **default behavior**

Like:

```
abstract class Animal {
    void eat() {
        System.out.println("Eating...");
    }
    abstract void sound();
}
class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}
```

Here, Dog gets eat() for free and only has to write sound()

❖ FullAbstractionDemo.java – ONE FILE, EVERYTHING INCLUDED

```
// 🔷 Interface with variables, constants, abstract, static & default methods
interface PaymentGateway {
    //  Constant (by default: public static final)
    String PROVIDER_NAME = "Universal Gateway";

    //  Abstract method (must be implemented)
    void pay(double amount);

    //  Default method (has body, can be overridden)
    default void greetUser() {
        System.out.println("Hello! Welcome to " + PROVIDER_NAME);
    }

    //  Static method (interface level only)
    static void showSupportedCurrencies() {
        System.out.println("Supported Currencies: INR, USD, EUR");
    }
}

// 🔷 Abstract class: common logic + constructor + abstract method
abstract class BasePaymentProcessor implements PaymentGateway {

    //  Instance variable
    protected String bankName;

    //  Constructor
    public BasePaymentProcessor(String bankName) {
        this.bankName = bankName;
    }
```

```
    // Common method (shared by all subclasses)
    public void connectToBankServer() {
        System.out.println(" Connecting to " + bankName + " bank server securely...");
    }

    // Abstract method (must be implemented)
    public abstract void pay(double amount);

    // Common method
    public void logTransaction(double amount) {
        System.out.println(" Logged transaction of ₹" + amount + " with " + bankName);
    }
}

// ◈ Final class: concrete implementation of everything
class RazorpayProcessor extends BasePaymentProcessor {

    // Constructor
    public RazorpayProcessor(String bankName) {
        super(bankName); // calling abstract class constructor
    }

    // Overriding abstract method
    @Override
    public void pay(double amount) {
        connectToBankServer(); // method from abstract class
        System.out.println(" Processing ₹" + amount + " via Razorpay...");
        logTransaction(amount); // method from abstract class
        System.out.println(" Payment Successful via Razorpay!\n");
    }

    // Overriding default method from interface (optional)
    @Override
    public void greetUser() {
        System.out.println(" Namaste! Razorpay welcomes you!");
    }
}

// ◈ Main class to test everything
public class FullAbstractionDemo {
    public static void main(String[] args) {

        // Interface reference — abstraction in action
        PaymentGateway payment = new RazorpayProcessor("HDFC");

        // Using default method (from interface, overridden in class)
        payment.greetUser();

        // Making payment — actual logic inside implementation class
        payment.pay(2500.00);

        // Static method of interface (called via interface name)
        PaymentGateway.showSupportedCurrencies();

        // Access constant
        System.out.println(" Payment Provider: " + PaymentGateway.PROVIDER_NAME);
    }
}
```

⇒ Module 6.8 – Encapsulation(private + getter + setter)

What is Encapsulation?

- **Binding data and methods together**
- **Hiding the internal details** using private variables
- Giving access via public **getters and setters**

◇ Real-Life Example to Simplify:

- Think of the main gate of your house:
- If the gate is directly open, anyone can enter (unauthorized access).
- But if there's a guard (setter) at the gate, the guard will check who's coming in (validation).
- Getters and setters act like that guard:

  - A **setter** ensures that whoever is entering (the data being set) is valid.
  - A **getter** ensures that only authorized people can view the data inside.

Encapsulation's "data hiding" doesn't mean that no one can access the data at all.
It means preventing direct access to the data and providing it in a controlled way.

In Java, we achieve encapsulation using:

1. Declare the variable of a class as a private (data hiding)
2. provide setter and getter method to modify and view the variable value (controlled access)

Example :

```
class House {

    //  Step 1: Private data (data hiding)
    private String currentGuest;
    private final String authorizedPassword = "welcome123";

    //  Step 2: Setter with validation (like a guard at gate)
    public void setGuest(String guestName, String password) {
        if (password.equals(authorizedPassword)) {
            this.currentGuest = guestName;
            System.out.println(guestName + " is allowed to enter the house.");
        } else {
            System.out.println("❌ Access denied for " + guestName + ". Invalid password!");
        }
    }

    // 🔎 Getter with protection (like asking guard who's inside)
    public String getGuest(String password) {
        if (password.equals(authorizedPassword)) {
            return "✅ Guest inside the house is: " + currentGuest;
        } else {
            return "❌ Access denied! Wrong password.";
        }
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        House myHouse = new House();

        // ❌ Trying to set a guest with wrong password
        myHouse.setGuest("John", "wrongpassword");

        // ✅ Setting a guest with the correct password
        myHouse.setGuest("Alice", "welcome123");

        // ❌ Trying to see the guest with wrong password
        System.out.println(myHouse.getGuest("wrongpassword"));

        // ✅ Viewing the guest with correct password
        System.out.println(myHouse.getGuest("welcome123"));
    }
}
```

⇒ Module 6.9 – Access Modifiers in Java

What Are Access Modifiers?

- Access modifiers **control the visibility** of classes, methods, and variables.
- They decide **"Who can see or access what?"** — just like privacy settings on your social media posts.

Real-Life Analogy: WhatsApp Privacy

Setting on WhatsApp	Java Access Modifier
"Only Me"	private
"My Contacts"	default
**"My Contacts + My Family"`	protected
"Everyone"	public

4 Types of Access Modifiers

Modifier	Same Class	Same Package	Subclass (Diff Package)	Outside Package
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
default	✅	✅	❌	❌
private	✅	❌	❌	❌

If no modifier is written, it's default by default.

Example:

```
// File: AccessModifierDemo.java

class Person {
    public String name = "Alice";          // 🔓 Public - accessible everywhere
    protected int age = 25;                // 🛡️ Protected - subclass & same package
    String city = "Mumbai";                // 🌐 Default - same package only
    private String password = "secret123"; // 🔒 Private - only inside this class
```

```
    public void showDetails() {
        System.out.println("Inside Person class:");
        System.out.println("Name: " + name);       // ✅
                System.out.println("Age: " + age);        // ✅
                System.out.println("City: " + city);      // ✅
                System.out.println("Password: " + password); // ✅
        }
}

class Student extends Person {
    public void showStudentInfo() {
        System.out.println("Inside Student (Child of Person):");
        System.out.println("Name: " + name);     // ✅ public - accessible
        System.out.println("Age: " + age);       // ✅ protected - accessible
        System.out.println("City: " + city);     // ✅ default - same package
        // System.out.println("Password: " + password); ❌ private - NOT accessible
    }
}

public class AccessModifierDemo {
    public static void main(String[] args) {
        Person p = new Person();
        Student s = new Student();

        System.out.println("== From Main (Same Package) ==");
        System.out.println("Name: " + p.name);   // ✅ public
        System.out.println("Age: " + p.age);     // ✅ protected (same package)
        System.out.println("City: " + p.city);   // ✅ default (same package)
        // System.out.println("Password: " + p.password); ❌ private

        System.out.println("\n== Calling from showDetails() ==");
        p.showDetails();

        System.out.println("\n== Calling from Student class ==");
        s.showStudentInfo();
    }
}
```


⇒ Module 6.10 – this and super Keyword in Java

✧ *This keyword is the reference variable that refer to the current object*

◆ **What is the need of this keyword?**

Without this, it would be difficult to distinguish between instance variables and local variables or method parameters that have the same name.

POINT TO REMEMBER :

1. **It can be used to invoke current class instance variable.**

CODE:

```
class Example1 {
```

```
    int number;

    Example1(int number) {
        this.number = number; // Using 'this' to distinguish instance variable from parameter
    }

    void display() {
        System.out.println("Number: " + this.number);
    }
}

public class Test1 {
    public static void main(String[] args) {
        Example1 obj = new Example1(10);
        obj.display();
    }
}
```

2. **<u>It can be used to invoke current class method.(Implicitly). -> if you don't use this keyword then compiler automatically add this keyword while invoking the method.</u>**

CODE:

```
class Example2 {
    void method1() {
        System.out.println("Method1 is called.");
        this.method2(); // Optional, as the compiler adds 'this' automatically
    }

    void method2() {
        System.out.println("Method2 is called.");
    }
}

public class Test2 {
    public static void main(String[] args) {
        Example2 obj = new Example2();
        obj.method1();
    }
}
```

3. **<u>This() can be used to invoke current class constructor</u>**

CODE:

```
class Example3 {
    Example3() {
        this(100); // Calling the parameterized constructor
        System.out.println("Default constructor called.");
    }

    Example3(int value) {
        System.out.println("Parameterized constructor called. Value: " + value);
    }
}

public class Test3 {
    public static void main(String[] args) {
```

```
        Example3 obj = new Example3();
    }
}
```

4. **<u>This can be used to pass an argument in the method call</u>**

CODE:

```
class Example4 {
    void method1(Example4 obj) {
        System.out.println("Method1 is called with object reference.");
    }

    void method2() {
        method1(this); // Passing the current instance
    }
}

public class Test4 {
    public static void main(String[] args) {
        Example4 obj = new Example4();
        obj.method2();
    }
}
```

*The issue occurs because you are using this inside the main() method, which is **static**. In Java, the this keyword is used as a reference to the current instance of the class. However, static methods, like main(), are not tied to any specific instance of the class—they belong to the class itself.*
Since this refers to an instance, it cannot be used in a static context where no instance exists.

5. **<u>This can be used to return the current class instance from a method.</u>**

CODE:

```
class Calculator {
    private double result;

    Calculator add(double value) {
        this.result += value; // Add the value
        return this;          // Return current instance
    }

    Calculator subtract(double value) {
        this.result -= value; // Subtract the value
        return this;          // Return current instance
    }

    Calculator multiply(double value) {
        this.result *= value; // Multiply by the value
        return this;          // Return current instance
    }

    Calculator divide(double value) {
        if (value != 0) {
            this.result /= value; // Divide by the value
        } else {
            System.out.println("Error: Division by zero is not allowed!");
        }
        return this; // Return current instance
```

```
    }

    void displayResult() {
        System.out.println("Final Result: " + this.result);
    }
}

public class Test {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Perform a series of calculations
        calc.add(10)
                .subtract(5)
                .multiply(4)
                .divide(2)
                .displayResult(); // Display the final result
    }
}
```

✧ **What is super in Java?**

*The super keyword is used **inside a subclass** to refer to the **parent (super) class**.*

Why Use super?

Purpose	Use Case
1 Call Parent Constructor	super()
2 Access Parent Variable	super.variableName
3 Call Parent Method	super.methodName()

Real-Life Analogy:

- Your dad (Parent class) has a car and rules.
- You (Child class) have your own bike and rules.
- But sometimes, you want to refer to your dad's car or rules – so you say:
- "Let me use my **super** dad's car" → super.car

Rules of super:

- super() must be the **first line** in the constructor if used.
- Only used in **inheritance** (child class).
- Can be used to:
  - Call **parent's constructor**.
  - Access **parent's method/field** if overridden.

Example:

```
class Person {
    String name = "John";

    Person() {
        System.out.println("Person constructor called");
    }
```

```
    void showDetails() {
        System.out.println("Person name: " + name);
    }
}

class Student extends Person {
    String name = "Alice";

    Student() {
        super(); // 1 Calling parent constructor
        System.out.println("Student constructor called");
    }

    void showDetails() {
        super.showDetails(); // 2 Calling parent method
        System.out.println("Student name: " + name);
        System.out.println("Parent name: " + super.name); // 3 Accessing parent variable
    }
}

public class SuperKeywordDemo {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println("---");
        s.showDetails();
    }
}
```


⇒ Module 6.11 – final keyword

✧ What is the final keyword?

*The final keyword in Java means **no more changes allowed**.*

Use With	Meaning
◆ final variable	◆ Value **cannot be changed** (constant)
◆ final method	◆ Method **cannot be overridden** in child class
◆ final class	◆ Class **cannot be inherited** (no subclass allowed)

Real-Life Analogy:

Scenario	Code
◆ Final Exam = You can't change your answers once submitted.	◆ final int marks = 100;
◆ Final Sealed Box = You can't put more things inside.	◆ final class ProductBox {}
◆ Final Rulebook = Kids cannot change the rules.	◆ final void rules() {}

1. final Variable – Constant Ban Gaya Bhai!

Syntax:

```
final int speed = 100;
```

```
speed = 200;  // ✕ Error: Cannot assign a value to final variable
```

Real Life Example:

- Final Exam – Once submitted, **you can't change** your answers.

Use Case:

- Constants like MAX_LIMIT, PI, API_KEY, etc.
- Immutable values.

2. final Method – Overriding Ki Bandhi Gadi

Syntax:

```
class Vehicle {
    final void start() {
        System.out.println("Vehicle starts");
    }
}
class Bike extends Vehicle {
    // void start() {} ✕ Error: Cannot override final method
}
```

Real Life Example:

- Final Rulebook – Kids **cannot change the rules** set by school.

Use Case:

- When you don't want your logic to be changed by subclasses.
- Useful in frameworks or core APIs.

3. final Class – No Further Inheritance Allowed

Syntax:

```
final class ProductBox {
    void show() {
        System.out.println("Product Box");
    }
}
// class CustomBox extends ProductBox {} ✕ Error
```

Real Life Example:

Final Sealed Box – You **can't add** anything inside it.

Use Case:

- When you want to prevent subclassing.
- Used in classes like java.lang.String, java.lang.Math.

CODE:

```
final class Hero {
    final int power = 100;

    final void showPower() {
        System.out.println("Power is " + power);
    }
```

```
}

class Player extends Hero {
    void showPower() {
        System.out.println("New Power");
    }
}
```

⇒ Module 6:12 : static in java - Think : "common for all":

The static keyword means: ***belonging to the class, not to objects.***

✧ static Variable – Shared Memory for All Objects

Why Use:

- To **save memory** – only one copy is created no matter how many objects exist.
- Useful when a value is **common to all instances** (e.g., interest rate, company name, school name, etc.)

Stored in:

Method Area of memory (once per class)

Example:

```
class Student {
    int roll;
    static String school = "ABC School";
}
```

Real-Life Analogy:

- All students have the **same school** – so why store "ABC School" again and again for each student? Just make it static.

Advantage:

- Memory Efficient 💾
- Easy to update for all objects at once

CODE :

✧ static Variable Example – "Shared Across All Objects"

```
class Student {
    int rollNo;
    static String schoolName = "ABC Public School";  // static variable

    Student(int rollNo) {
        this.rollNo = rollNo;
    }

    void show() {
        System.out.println("Roll No: " + rollNo + ", School: " + schoolName);
    }

    public static void main(String[] args) {
        Student s1 = new Student(101);
        Student s2 = new Student(102);

        s1.show();
```

```
        s2.show();

        // Changing schoolName via class
        Student.schoolName = "XYZ International School";

        s1.show();
        s2.show();  // School name changed for both
    }
}
```

✧ static Method – Call Without Object

Why Use:

- If the method **does not depend on instance variables**, you should make it static.
- Can be called using class name → no need to create object = **fast and memory-saving**

Stored in:

Method Area

Example:

```
class Calculator {
    static int square(int x) {
        return x * x;
    }
}
Usage:

int sq = Calculator.square(5);
```

Real-Life Analogy:

Like a **common calculator app** — use it without creating a new calculator every time.

Advantage:

- Save memory: No object needed.
- Utility methods like Math.max(), Collections.sort() are static.

CODE:

✧ static Method Example – "No Object Required"

```
class MathUtils {

    static int square(int n) {
        return n * n;
    }

    static int cube(int n) {
        return n * n * n;
    }

    public static void main(String[] args) {
        System.out.println("Square of 5: " + MathUtils.square(5));
        System.out.println("Cube of 3: " + MathUtils.cube(3));
    }
}
```

✧ static Block – Runs Only Once When Class Loads

Why Use:

- To initialize **static variables** or code only once during class loading.
- Runs **before constructor**, **before main()**, **before any method**

Stored in:

Method Area, executed by ClassLoader

Example:

```
class App {
    static int version;

    static {
        version = 1;
        System.out.println("App version initialized");
    }
}
```

Real-Life Analogy:

- Like a **theater setup before show starts** — run once to prepare everything before users (objects) come.

Advantage:

- One-time setup.
- Useful for **config loading**, **DB connections**, **static resource initialization**.

CODE:

✧ static Block Example – "Runs Once When Class is Loaded"

```
class AppInitializer {

    static {
        System.out.println("Static Block: App is initializing...");
        config = "Production Config Loaded";
    }

    static String config;

    public static void main(String[] args) {
        System.out.println("Main Method: App Started");
        System.out.println("Config: " + config);
    }
}
```


⇒ Module 6.13 – Inner Classes ka number:

✧ What is an Inner Class?

- A class defined **inside another class** is called an **inner class**.

Why?

- For **logical grouping** of classes.

- For better **encapsulation**.
- Useful when a class is **used only by one outer class**.

1. Normal Inner Class (Non-static)

Meaning:

- Can **access all members** (even private) of outer class.
- Needs an **object of outer class** to be created.

Example:

```
class Outer {
    private String msg = "Hello from Outer";
    class Inner {
        void display() {
            System.out.println("Inner accessing: " + msg);
        }
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();  // syntax
        inner.display();
    }
}
```

Real-life Analogy:

- *Like a **classroom** inside a **school** — it belongs to it, needs it to exist.*

2. Static Nested Class

Meaning:

- Declared as static.
- Can access **only static members** of outer class.
- Doesn't need outer object to be created.

Example:

```
class Container {
    static int value = 50;
    static class Nested {
        void show() {
            System.out.println("Static Nested accessing: " + value);
        }
    }
    public static void main(String[] args) {
        Container.Nested obj = new Container.Nested();  // no outer object needed
        obj.show();
    }
}
```

Real-life Analogy:

- *Like a **static nested folder** inside a project — it can exist without opening the outer folder.*

3. Anonymous Inner Class

Meaning:

- A class **without a name**, created to override or implement methods on the spot.

- Commonly used with **interfaces or abstract classes**.
- Best used when you want to use a class only once.

Example:

```
abstract class Animal {
   abstract void makeSound();
}
public class Zoo {
   public static void main(String[] args) {
      Animal tiger = new Animal() {
         void makeSound() {
            System.out.println("Tiger says: Roar!");
         }
      };
      tiger.makeSound();
   }
}
```

Real-life Analogy:

- *Like a **temporary performer** in a show — no permanent name, just performs and goes!*

⇒ Module 6.14 – Interview Questions & Assignment

Beginner Level

No.	Question	Answer
1	◆ What is a class in Java?	◆ A blueprint for creating objects.
2	◆ What is an object?	◆ A real-world instance of a class.
3	◆ What is a constructor?	◆ A special method used to initialize objects.
4	◆ What is the default constructor?	◆ A constructor with no parameters, created by Java if none defined.
5	◆ What is this keyword?	◆ Refers to the current object.
6	◆ What is inheritance?	◆ When one class inherits features from another.
7	◆ What is method overriding?	◆ Providing a new definition of a method in child class.
8	◆ What is polymorphism?	◆ One thing behaving in many forms (method overloading/overriding).
9	◆ What is encapsulation?	◆ Binding data and methods together, using private fields + public getters/setters.
10	◆ What is abstraction?	◆ Hiding implementation details and showing only essential features.

Intermediate Level

No.	Question	Answer
1	◆ What are the types of inheritance in Java?	◆ Single, Multilevel, Hierarchical
2	◆ What is constructor chaining?	◆ Calling one constructor from another using this() or super()
3	◆ Can we override a static method?	◆ No, static methods can't be overridden.
4	◆ Can a class be both abstract and final?	◆ No, it's a contradiction.
5	◆ Can we create an object of an abstract	◆ No, but we can reference it.

	class?	
6	◆ What is the use of super keyword?	◆ To access parent class constructor, method, or variable.
7	◆ Can private methods be inherited?	◆ No, private members are not inherited.
8	◆ What is the default access modifier in Java?	◆ Package-private (no modifier).
9	◆ Can we overload constructors?	◆ Yes, by using different parameter lists.
10	◆ What is the use of static block?	◆ To run one-time code before any object or method is accessed.

Expert Level

No.	Question	Answer
1	◆ Why is the String class final?	◆ To ensure immutability and prevent subclass-based security issues.
2	◆ What is an anonymous inner class?	◆ A class without a name, created to override/implement on the spot.
3	◆ What is the difference between abstract class and interface?	◆ Abstract class can have both abstract and non-abstract methods, interface only abstract (Java 8+ has default methods).
4	◆ Can we declare a constructor in an interface?	◆ No, interfaces cannot have constructors.
5	◆ What is method signature in overriding?	◆ Method name + parameter list (return type doesn't matter).
6	◆ What is the purpose of access modifiers in OOP?	◆ To control visibility and protect data.
7	◆ How does JVM resolve overridden methods at runtime?	◆ Through dynamic method dispatch (runtime polymorphism).
8	◆ What happens if we try to extend a final class?	◆ Compile-time error – final classes can't be extended.
9	◆ When should you use static nested class over inner class?	◆ When the inner class doesn't need outer class's non-static members.
10	◆ What are the four pillars of OOP?	◆ Inheritance, Polymorphism, Abstraction, Encapsulation.